

Lexical Analyzer



Siveen Said Sayed Ahmed 21P0149

Hamsa Ahmed Abdelmageed 20P1874

Malak Mohamed Salem Elgeziry 21P0277

Hager Hesham Mohamed Elsawy 21P0297

Mariam Mahmoud Mohamed Abdou 21P0214

March 7, 2024

Table of Contents

[Introduction 3](#_Toc162540045)

[1) Code 5](#_Toc162540046)

[1.1) Include 5](#_Toc162540047)

[1.2) Token 5](#_Toc162540048)

[1.3) Symbol Table 6](#_Toc162540049)

[1.4) Read from File 7](#_Toc162540050)

[1.5) Get token type 7](#_Toc162540051)

[1.6) getNextToken 8](#_Toc162540052)

[1.7) Do  9](#_Toc162540053)

[1.8) Switch case 10](#_Toc162540054)

[Case 1 10](#_Toc162540055)

[Case 2 and 3 11](#_Toc162540056)

[Case 4 and 5  11](#_Toc162540057)

[Case 6 and 7 12](#_Toc162540058)

[Case 8 12](#_Toc162540059)

[Case 9 12](#_Toc162540060)

[Case 10 12](#_Toc162540061)

[Case 11 12](#_Toc162540062)

[Case 12 12](#_Toc162540063)

[Case 13 13](#_Toc162540064)

[Case 14 13](#_Toc162540065)

[Case 15 13](#_Toc162540066)

[Case 16 13](#_Toc162540067)

[Case 17 13](#_Toc162540068)

[Case 18 13](#_Toc162540069)

[Case 19 13](#_Toc162540070)

[Case 20 13](#_Toc162540071)

[Case 21 14](#_Toc162540072)

[Case 22 14](#_Toc162540073)

[Case 23 14](#_Toc162540074)

[Case 24 14](#_Toc162540075)

[Case 26 14](#_Toc162540076)

[Case 27 14](#_Toc162540077)

[Case 28 15](#_Toc162540078)

[Case 29 15](#_Toc162540079)

[Case 30 15](#_Toc162540080)

[Case 31 15](#_Toc162540081)

[Case 32 15](#_Toc162540082)

[Case 33 15](#_Toc162540083)

[Last 2 functions 16](#_Toc162540084)

# Introduction

In the realm of computer programming, complex lines of code serve as the building blocks for powerful software applications. However, before a computer can execute these instructions, it must first unravel the intricacies of human-readable source code and translate it into a language it can comprehend. This is where lexical analysis, a fundamental aspect of the compilation process, comes into play.

Lexical analysis, also known as scanning, is the initial phase of the compiler that aims to understand the structure and meaning of the source code. It involves breaking down the text-based representation of a program into smaller meaningful units called tokens. These tokens act as the vocabulary of the programming language, representing keywords, identifiers, literals, operators, and other syntactic elements.

During the lexical analysis phase, a program's source code is scanned character by character. The scanner, also known as the Lexer, employs a set of rules defined by the programming language's specification to recognize and categorize the characters into different token types. These rules, often expressed as regular expressions, define the patterns that identify keywords, identifiers, literals, and other language-specific constructs.

Tokens serve as the atomic units of a programming language, representing meaningful elements that contribute to the overall structure and semantics of the code. For example, in the C++ programming language, tokens can include keywords like "if" or "while" variable names like "count" or "average," numeric literals such as "42" or "-3.14," and various operators like "+," "-", "\*", and "/".

As the lexical analyzer scans the source code, it emits a stream of tokens, each carrying valuable information about its type and value. These tokens form the foundation for subsequent compilation stages, such as parsing, semantic analysis, and code generation. By understanding the lexical structure of the program, the compiler gains insights into its syntax and can detect errors or inconsistencies early on.

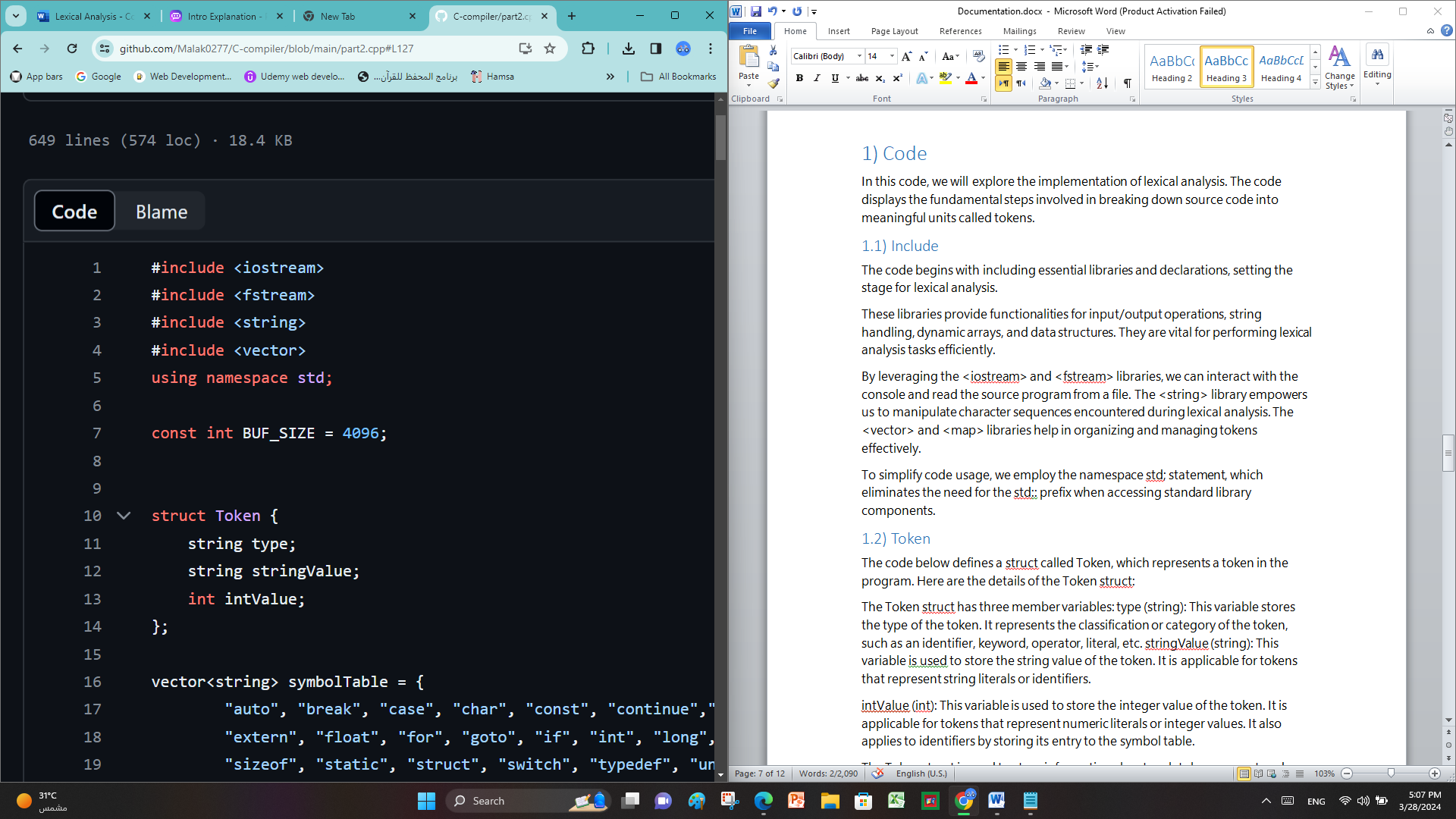
Moreover, lexical analysis facilitates the creation of a symbol table, a data structure that records information about variables, functions, and other identifiers found in the code. The symbol table acts as a reference for the compiler, providing a centralized repository for storing and retrieving information about the program's symbols during later stages of compilation.

Lexical analysis plays a critical role in the broader process of transforming human-readable code into machine-executable instructions. By dissecting source code into tokens and establishing a symbol table, lexical analysis lays the groundwork for subsequent stages to analyze, interpret, and transform code into a form that can be executed by a computer.

# 1) Code

In this code, we will explore the implementation of lexical analysis. The code displays the fundamental steps involved in breaking down source code into meaningful units called tokens.

## 1.1) Include

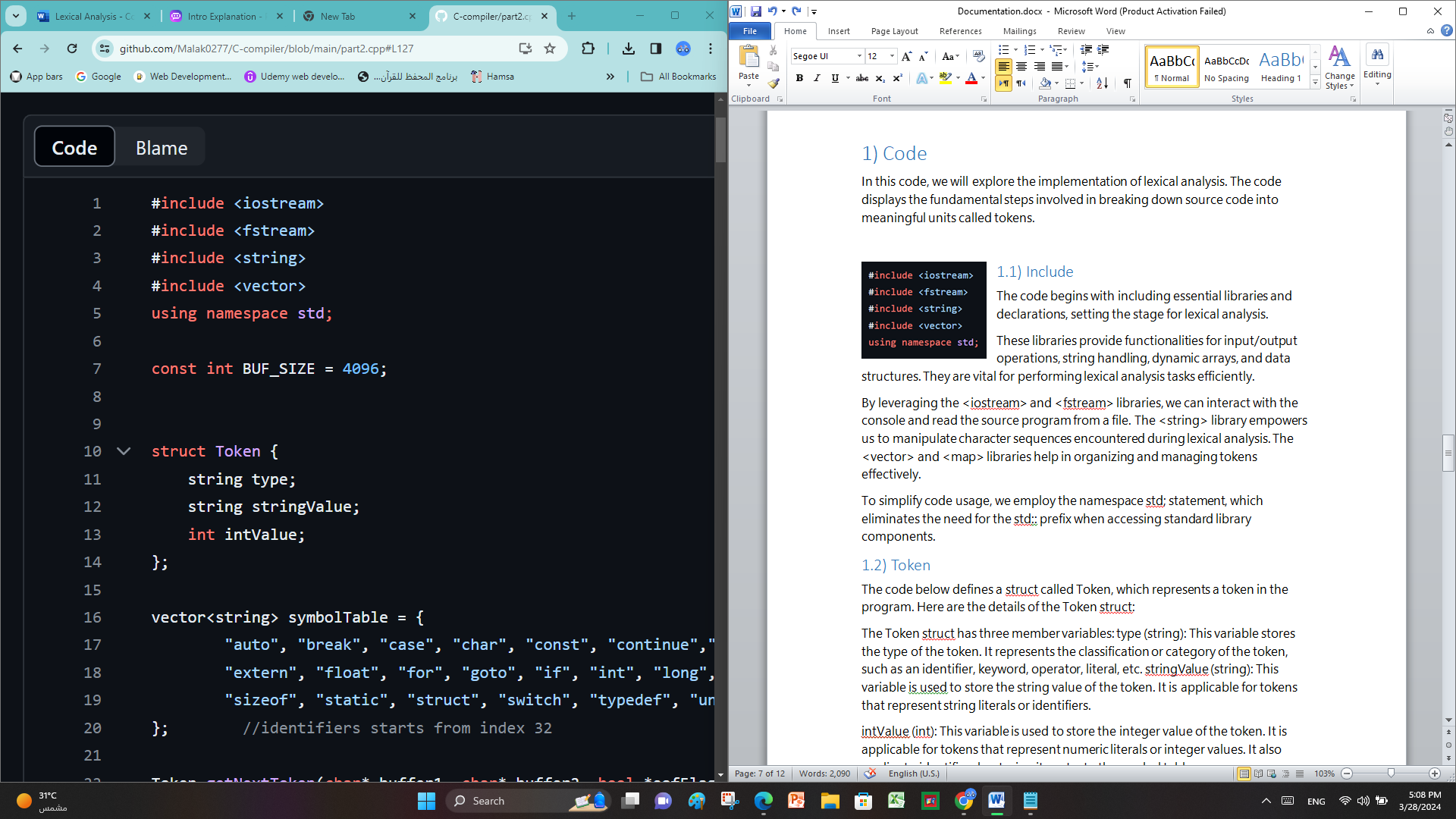
The code begins with including essential libraries and declarations, setting the stage for lexical analysis.

These libraries provide functionalities for input/output operations, string handling, dynamic arrays, and data structures. They are vital for performing lexical analysis tasks efficiently.

By leveraging the <iostream> and <fstream> libraries, we can interact with the console and read the source program from a file. The <string> library empowers us to manipulate character sequences encountered during lexical analysis. The <vector> and <map> libraries help in organizing and managing tokens effectively.

To simplify code usage, we employ the namespace std; statement, which eliminates the need for the std:: prefix when accessing standard library components.

## 1.2) Token

The code below defines a struct called Token, which represents a token in the program. Here are the details of the Token struct:

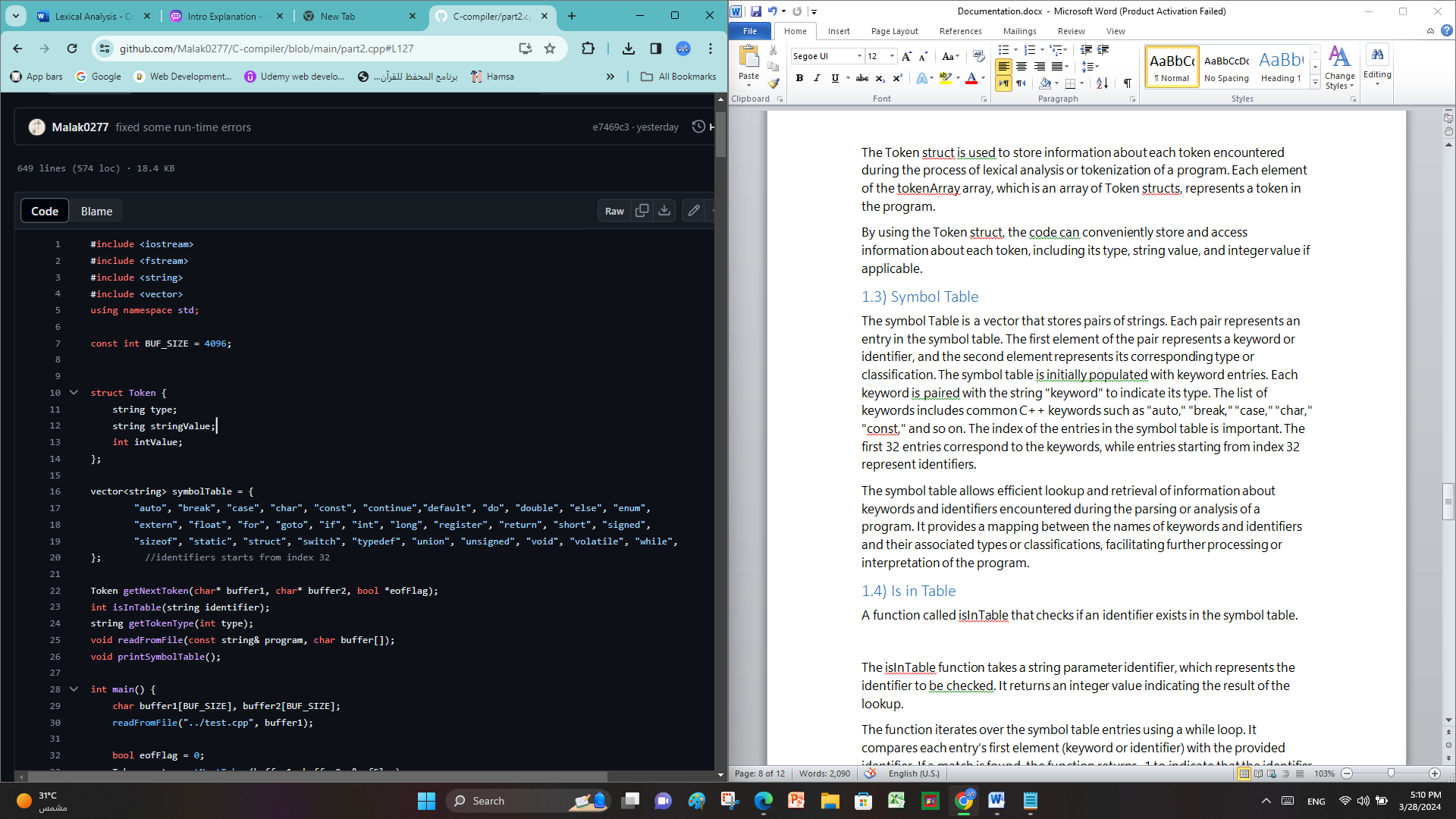
The Token struct has three member variables: type (string): This variable stores the type of the token. It represents the classification or category of the token, such as an identifier, keyword, operator, literal, etc. stringValue (string): This variable is used to store the string value of the token. It is applicable for tokens that represent string literals or identifiers.

intValue (int): This variable is used to store the integer value of the token. It is applicable for tokens that represent numeric literals or integer values. It also applies to identifiers by storing its entry to the symbol table.

The Token struct is used to store information about each token encountered during the process of lexical analysis or tokenization of a program. Each element of the tokenArray array, which is an array of Token structs, represents a token in the program.

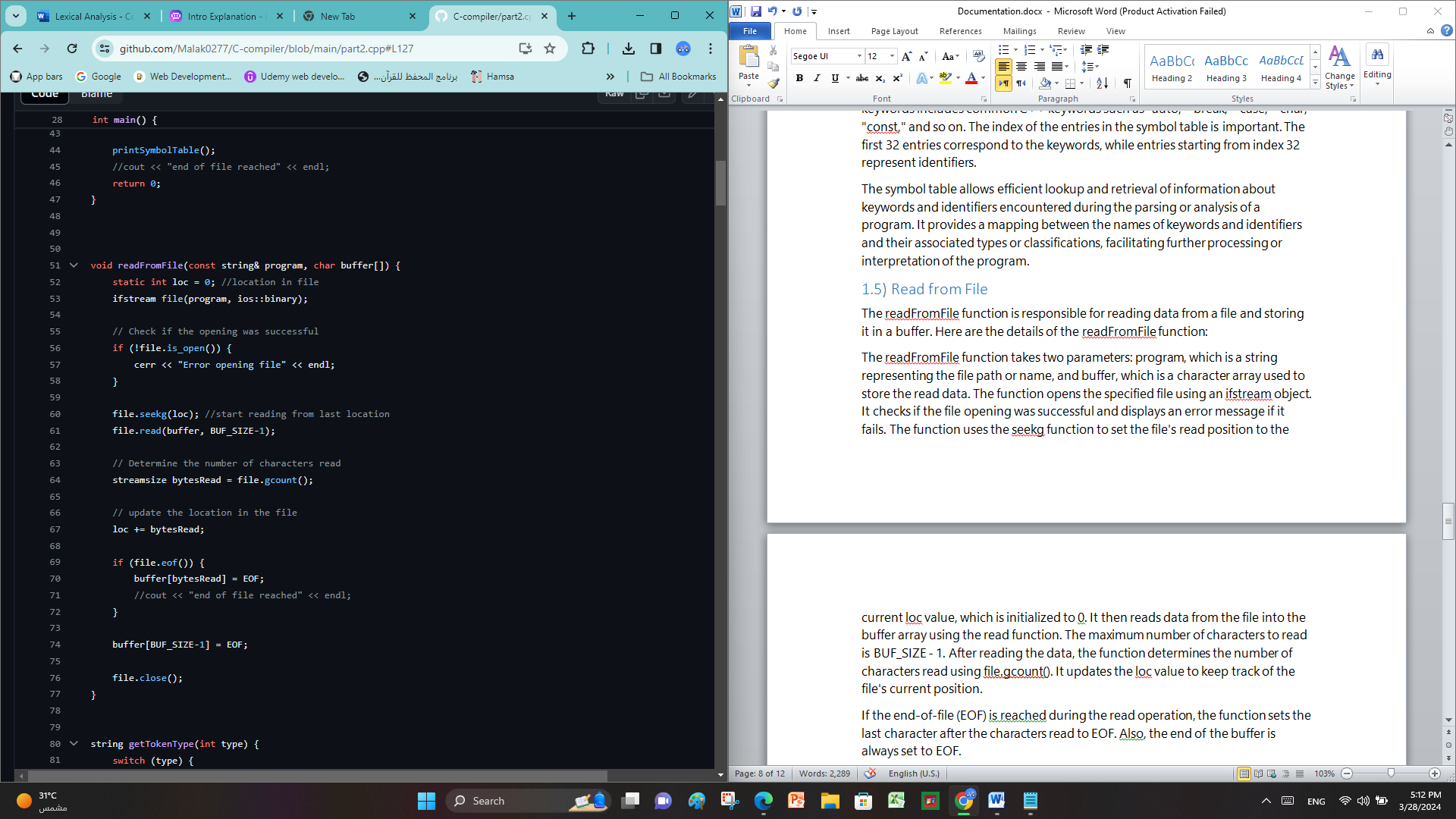
By using the Token struct, the code can conveniently store and access information about each token, including its type, string value, and integer value if applicable.

## 1.3) Symbol Table

The symbol Table is a vector that stores pairs of strings. Each pair represents an entry in the symbol table. The first element of the pair represents a keyword or identifier, and the second element represents its corresponding type or classification. The symbol table is initially populated with keyword entries. Each keyword is paired with the string "keyword" to indicate its type. The list of keywords includes common C++ keywords such as "auto," "break," "case," "char," "const," and so on. The index of the entries in the symbol table is important. The first 32 entries correspond to the keywords, while entries starting from index 32 represent identifiers.

The symbol table allows efficient lookup and retrieval of information about keywords and identifiers encountered during the parsing or analysis of a program. It provides a mapping between the names of keywords and identifiers and their associated types or classifications, facilitating further processing or interpretation of the program.

## 1.4) Read from File

The readFromFile function is responsible for reading data from a file and storing it in a buffer. Here are the details of the readFromFile function:

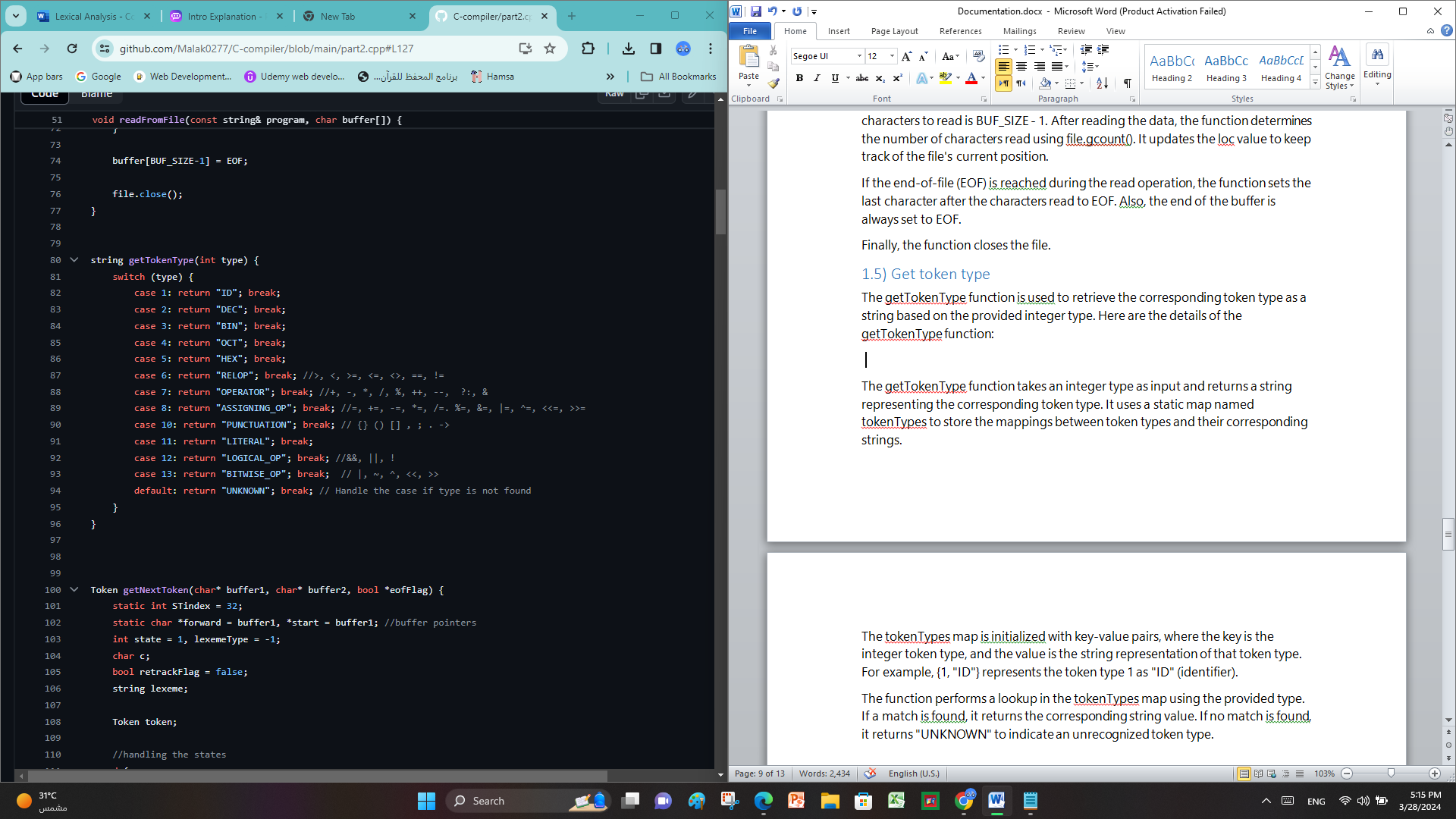
The readFromFile function takes two parameters: program, which is a string representing the file path or name, and buffer, which is a character array used to store the read data. The function opens the specified file using an ifstream object. It checks if the file opening was successful and displays an error message if it fails. The function uses the seekg function to set the file's read position to the current loc value, which is initialized to 0. It then reads data from the file into the buffer array using the read function. The maximum number of characters to read is BUF\_SIZE - 1. After reading the data, the function determines the number of characters read using file.gcount(). It updates the loc value to keep track of the file's current position.

If the end-of-file (EOF) is reached during the read operation, the function sets the last character after the characters read to EOF. Also, the end of the buffer is always set to EOF.

Finally, the function closes the file.

## 1.5) Get token type

The getTokenType function is used to retrieve the corresponding token type as a string based on the provided integer type. Here are the details of the getTokenType function:



The getTokenType function takes an integer type as input and returns a string representing the corresponding token type. It uses a static map named tokenTypes to store the mappings between token types and their corresponding strings.

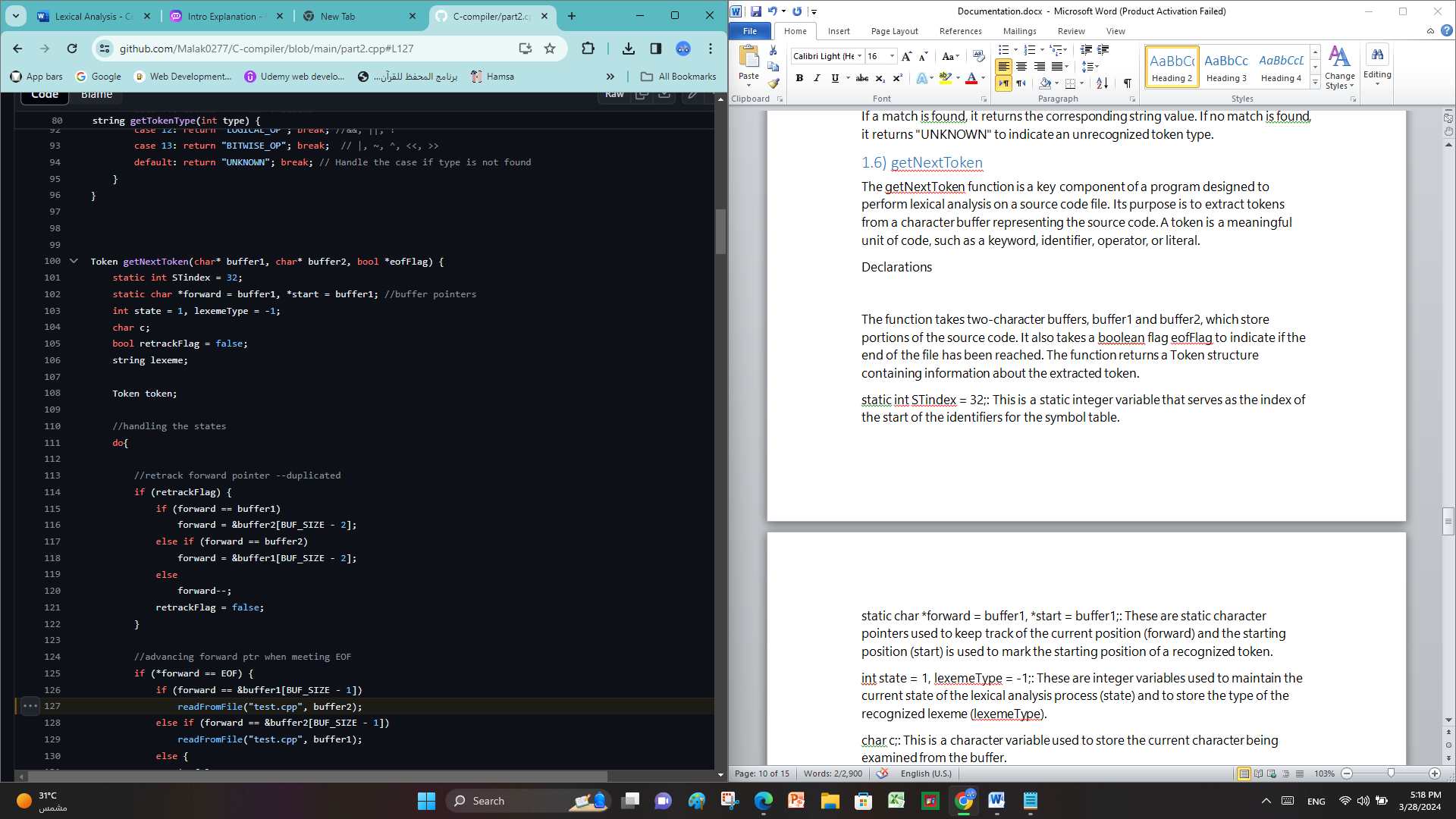
The tokenTypes map is initialized with key-value pairs, where the key is the integer token type, and the value is the string representation of that token type. For example, {1, "ID"} represents the token type 1 as "ID" (identifier).

The function performs a lookup in the tokenTypes map using the provided type. If a match is found, it returns the corresponding string value. If no match is found, it returns "UNKNOWN" to indicate an unrecognized token type.

## 1.6) getNextToken

The getNextToken function is a key component of a program designed to perform lexical analysis on a source code file. Its purpose is to extract tokens from a character buffer representing the source code. A token is a meaningful unit of code, such as a keyword, identifier, operator, or literal.

Declarations



The function takes two-character buffers, buffer1 and buffer2, which store portions of the source code. It also takes a boolean flag eofFlag to indicate if the end of the file has been reached. The function returns a Token structure containing information about the extracted token.

static int STindex = 32;: This is a static integer variable that serves as the index of the start of the identifiers for the symbol table.

static char \*forward = buffer1, \*start = buffer1;: These are static character pointers used to keep track of the current position (forward) and the starting position (start) is used to mark the starting position of a recognized token.

int state = 1, lexemeType = -1;: These are integer variables used to maintain the current state of the lexical analysis process (state) and to store the type of the recognized lexeme (lexemeType).

char c;: This is a character variable used to store the current character being examined from the buffer.

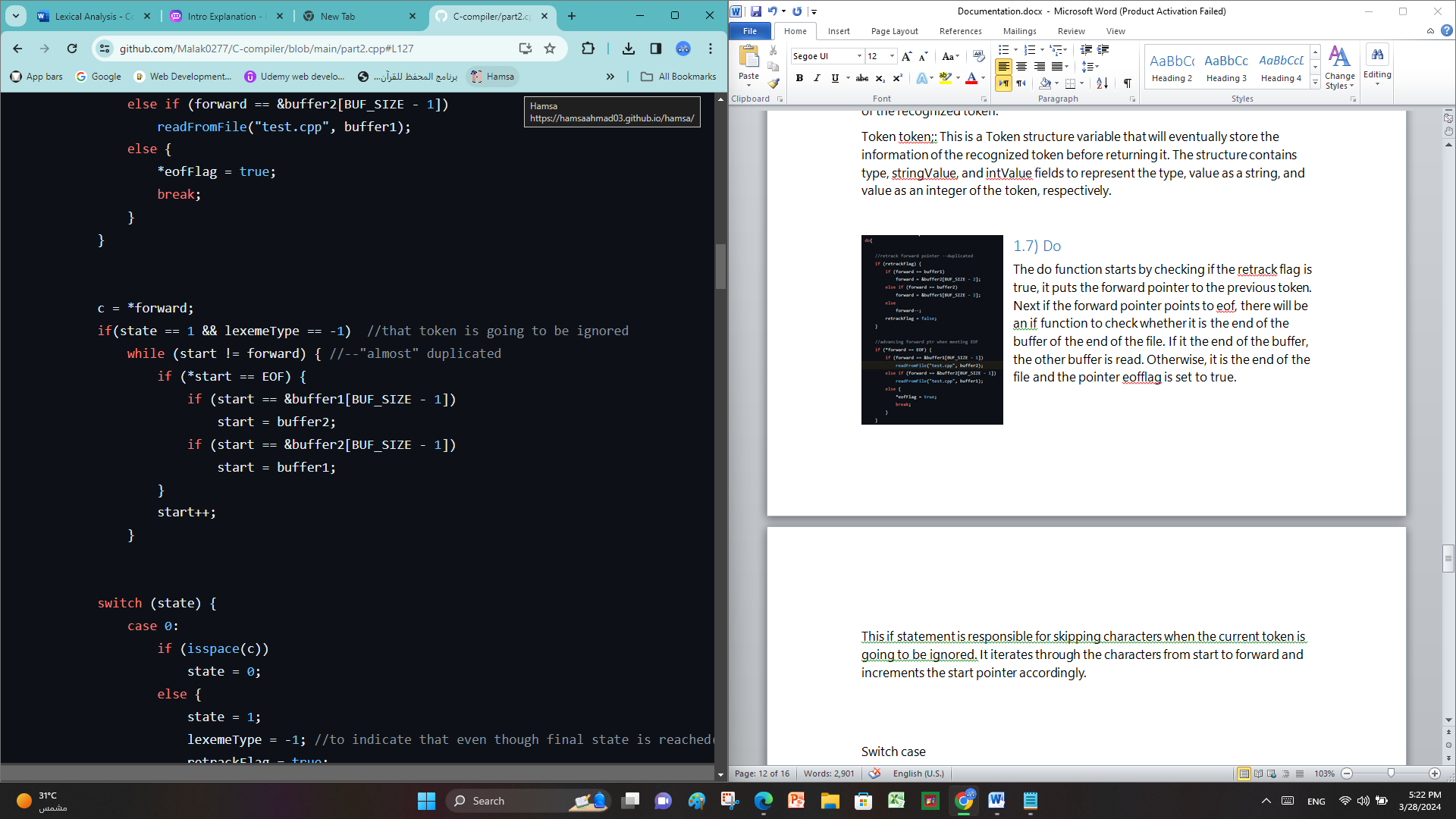
bool retrackFlag = false;: This boolean flag is used to indicate whether the forward pointer needs to be retracted or moved back to the previous position. It is initially set to false.

string lexeme;: This is a string variable used to store the lexeme (the actual value) of the recognized token.

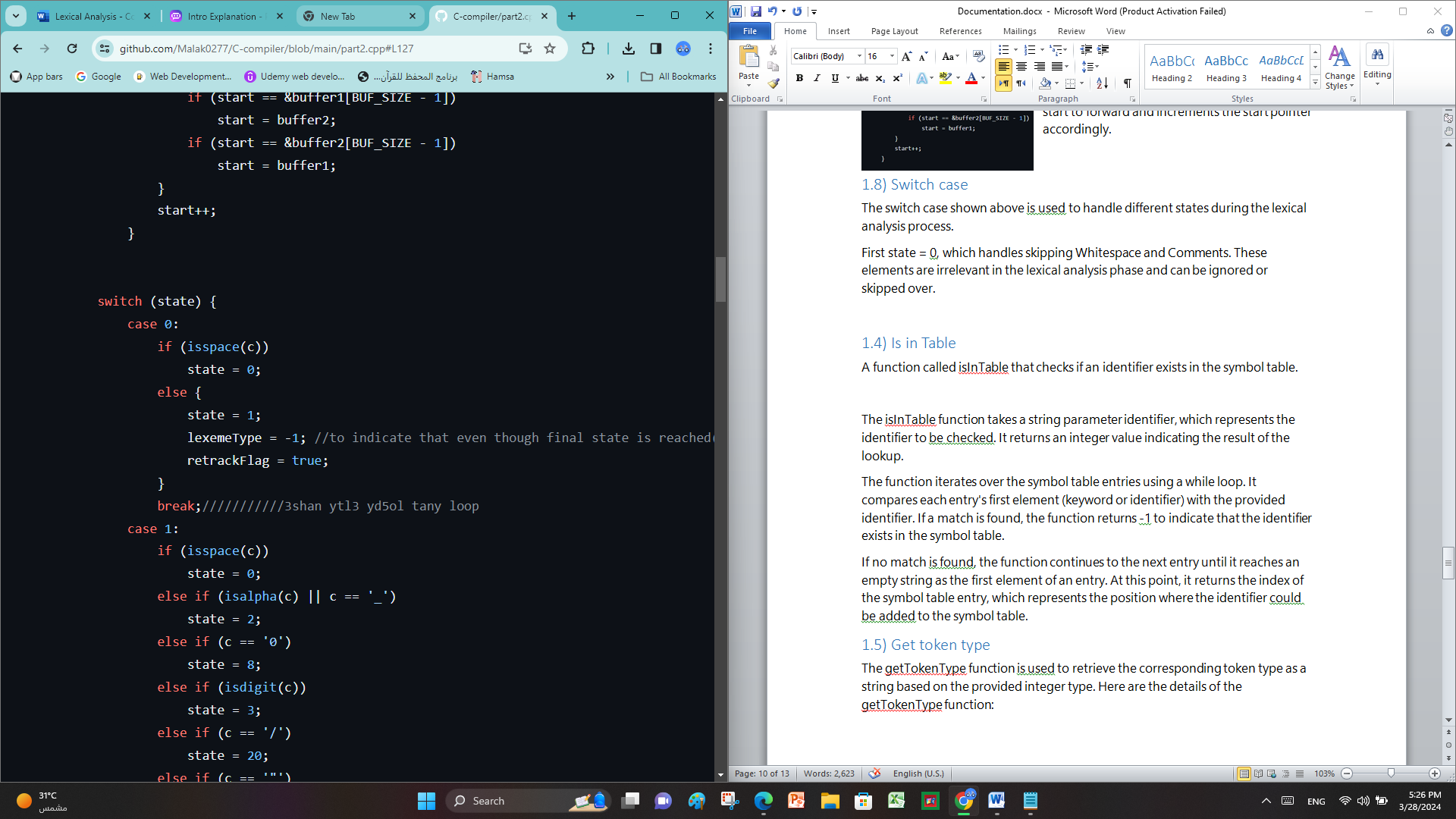
Token token;: This is a Token structure variable that will eventually store the information of the recognized token before returning it. The structure contains type, stringValue, and intValue fields to represent the type, value as a string, and value as an integer of the token, respectively.

## 1.7) Do

The do function starts by checking if the retrack flag is true, it puts the forward pointer to the previous token. Next if the forward pointer points to eof, there will be an if function to check whether it is the end of the buffer of the end of the file. If it the end of the buffer, the other buffer is read. Otherwise, it is the end of the file and the pointer eofflag is set to true.

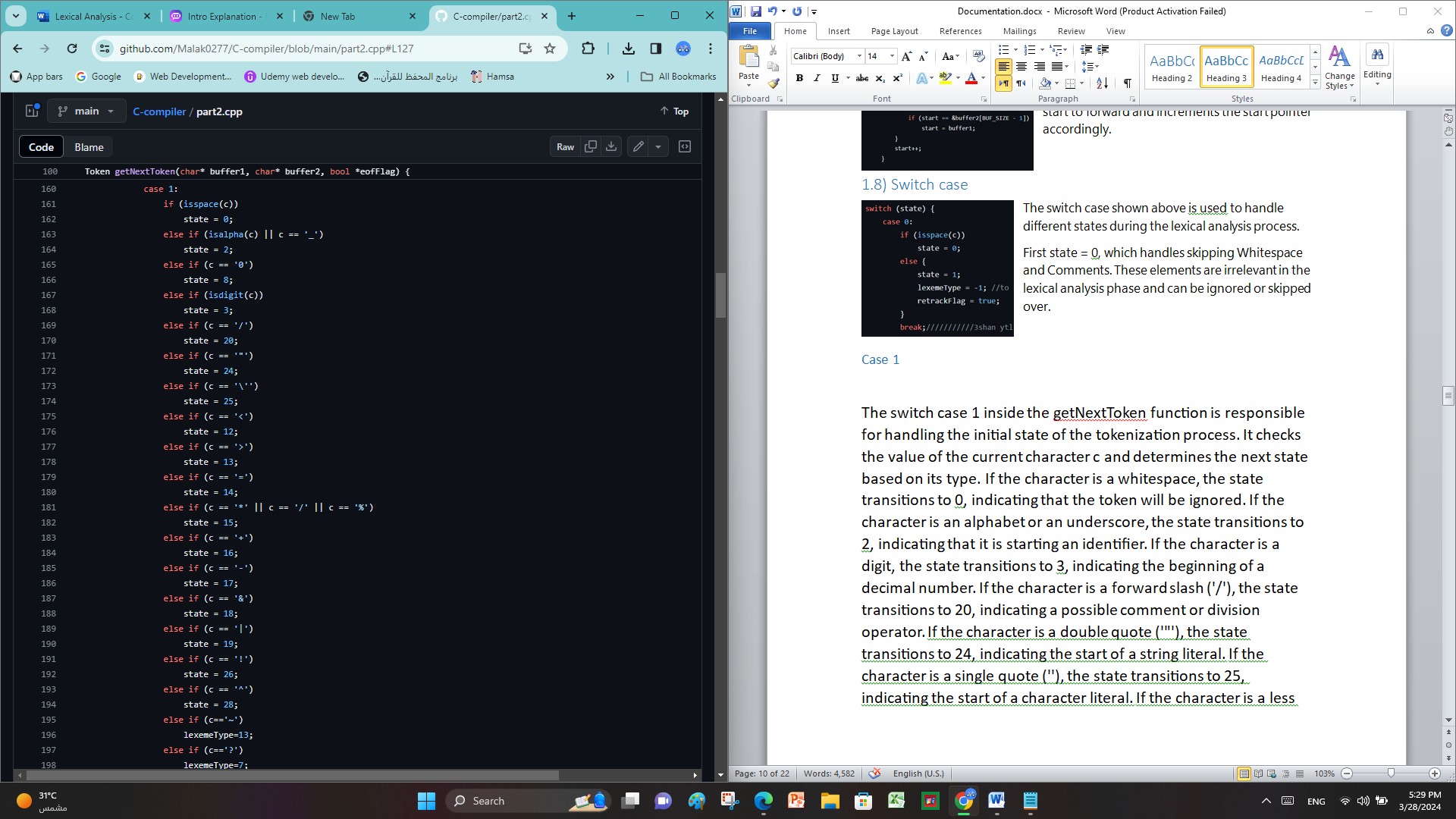
This if statement is responsible for skipping characters when the current token is going to be ignored. It iterates through the characters from start to forward and increments the start pointer accordingly.

## 1.8) Switch case

The switch case shown above is used to handle different states during the lexical analysis process.

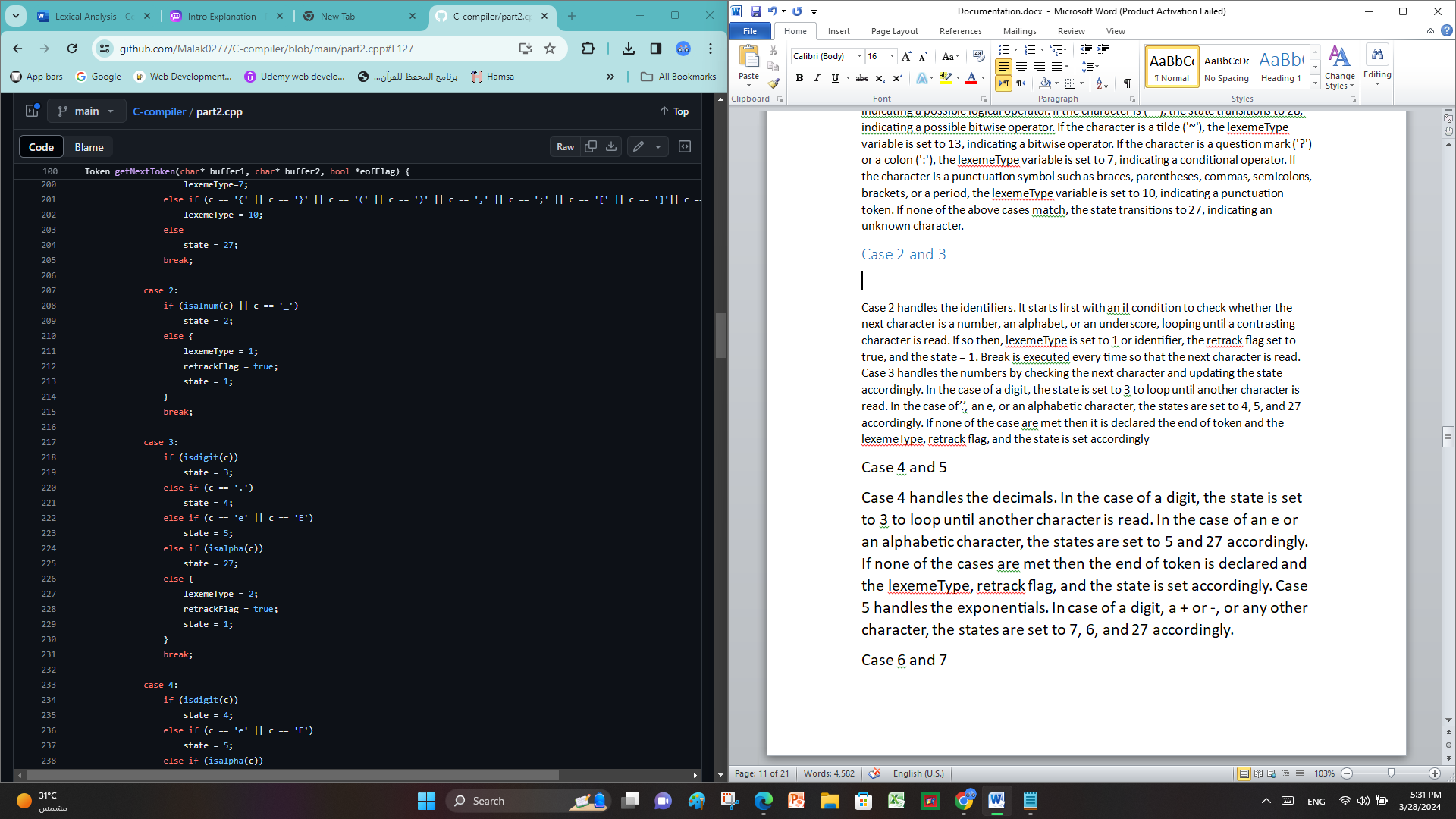
First state = 0, which handles skipping Whitespace and Comments. These elements are irrelevant in the lexical analysis phase and can be ignored or skipped over.

### Case 1

The switch case 1 inside the getNextToken function is responsible for handling the initial state of the tokenization process. It checks the value of the current character c and determines the next state based on its type. If the character is a whitespace, the state transitions to 0, indicating that the token will be ignored. If the character is an alphabet or an underscore, the state transitions to 2, indicating that it is starting an identifier. If the character is a digit, the state transitions to 3, indicating the beginning of a decimal number. If the character is a forward slash ('/'), the state transitions to 20, indicating a possible comment or division operator. If the character is a double quote ('"'), the state transitions to 24, indicating the start of a string literal. If the character is a single quote (''), the state transitions to 25, indicating the start of a character literal. If the character is a less than ('<'), the state transitions to 12, indicating a possible relational operator. If the character is a greater than ('>'), the state transitions to 13, indicating a possible relational operator. If the character is an equal sign ('='), the state transitions to 14, indicating an assignment operator. If the character is an asterisk ('\*'), a forward slash ('/'), or a percent sign ('%'), the state transitions to 15, indicating a possible arithmetic operator.

If the character is a plus sign ('+'), the state transitions to 16, indicating a possible arithmetic operator. If the character is a minus sign ('-'), the state transitions to 17, indicating a possible arithmetic operator. If the character is an ampersand ('&'), the state transitions to 18, indicating a possible logical or bitwise operator. If the character is a vertical bar ('|'), the state transitions to 19, indicating a possible logical or bitwise operator. If the character is an exclamation mark ('!'), the state transitions to 26, indicating a possible logical operator. If the character is ('^'), the state transitions to 28, indicating a possible bitwise operator. If the character is a tilde ('~'), the lexemeType variable is set to 13, indicating a bitwise operator. If the character is a question mark ('?') or a colon (':'), the lexemeType variable is set to 7, indicating a conditional operator. If the character is a punctuation symbol such as braces, parentheses, commas, semicolons, brackets, or a period, the lexemeType variable is set to 10, indicating a punctuation token. If none of the above cases match, the state transitions to 27, indicating an unknown character.

### Case 2 and 3

Case 2 handles the identifiers. It starts first with an if condition to check whether the next character is a number, an alphabet, or an underscore, looping until a contrasting character is read. If so then, lexemeType is set to 1 or identifier, the retrack flag set to true, and the state = 1. Break is executed every time so that the next character is read. Case 3 handles the numbers by checking the next character and updating the state accordingly. In the case of a digit, the state is set to 3 to loop until another character is read. In the case of’.’, an e, or an alphabetic character, the states are set to 4, 5, and 27 accordingly. If none of the case are met then it is declared the end of token and the lexemeType, retrack flag, and the state is set accordingly

### Case 4 and 5

Case 4 handles the decimals. In the case of a digit, the state is set to 3 to loop until another character is read. In the case of an e or an alphabetic character, the states are set to 5 and 27 accordingly. If none of the cases are met then the end of token is declared and the lexemeType, retrack flag, and the state is set accordingly. Case 5 handles the exponentials. In case of a digit, a + or -, or any other character, the states are set to 7, 6, and 27 accordingly.

### Case 6 and 7

Case 6 handles the positive and negatives in the exponentials. In case of a digit, the state is set to 7. Otherwise, the state is set to 27 or error. Case 7 handles the characters written after the sign in the exponentials. In case of a digit, the state is set to 7, looping until the numbers end. In the case of an alphabet, the state is set to 27 or error. Otherwise, a token has been found and the lexemeType, retrack flag, and the state is set accordingly.

### Case 8

Case 8 handles the binary, hexadecimal, and octal. In the case of b, x, digit, or an alphabet, the states are set to 9, 10, 11, and 27 accordingly. Otherwise, a token has been found and the lexemeType, retrack flag, and the state is set accordingly.

### Case 9

Case 9 handles the binary number. An if statement is made to check if the numbers read is 0 or 1. If the condition is satisfied, then the state is set to 9 looping until another character is read. Else if check if the character is alnum, setting the state to 27 or error. Otherwise, a token has been found and the lexemeType, retrack flag, and the state is set accordingly.

### Case 10

Case 9 handles the binary number. An if statement is made to check if the numbers read is part of the hexadecimal numbers. If the condition is satisfied, then the state is set to 10 looping until another character is read. Else if checks if the character is alphabetic, the state is set to 27 or error. Otherwise, a token has been found and the lexemeType, retrack flag, and the state is set accordingly.

### Case 11

Case 11 handles the octal number. An if statement is made to check if the numbers read is not a part of the octal numbers. If the condition is satisfied, then the state is set to 27 or error. Else if checks if the character is a number between 0 and 7, the state is set to 11, looping until another character is found. If an alphabet is read, then the state is set to 27 or error. Otherwise, a token has been found and the lexemeType, retrack flag, and the state is set accordingly.

### Case 12

Case 12: This case corresponds to the character '<'. It is responsible for handling the possibility of a relational operator. Depending on the next character, it can form relational operators such as '<', '<=', or '<<'.

### Case 13

Case 13: This case corresponds to the character '>'. Like case 12, it handles the possibility of a relational operator. Depending on the next character, it can form relational operators such as '>', '>=', or '>>'.

### Case 14

Case 14: This case corresponds to the character '='. It is responsible for handling the possibility of an assignment operator. Depending on the next character, it can form assignment operators such as '=', '+=', '-=', '\*=', or '/='.

### Case 15

Case 15: This case corresponds to the characters '\*', '/', or '%'. It handles the possibility of arithmetic operators.

### Case 16

Case 16: This case corresponds to the character '+'. It is responsible for handling the possibility of an arithmetic operator. Depending on the next character, it can form arithmetic operators such as '+', '++', or '+='.

### Case 17

Case 17: This case corresponds to the character '-'. It is responsible for handling the possibility of an arithmetic operator. Depending on the next character, it can form arithmetic operators such as '-', '--', or '-='.

### Case 18

Case 18: This case corresponds to the character '&'. It is responsible for handling the possibility of a logical or bitwise operator. Depending on the next character, it can form operators such as '&&', '&=', or bitwise operations.

### Case 19

Case 19: This case corresponds to the character '|'. It is responsible for handling the possibility of a logical or bitwise operator. Depending on the next character, it can form operators such as '||', '|=', or bitwise operations.

### Case 20

Case 20: This case corresponds to the character '/' inside the getNextToken function. It is responsible for handling the possibility of a comment or division operator. Depending on the next character, it can transition to different states to handle single-line or multi-line comments, or it can recognize a division operator.

### Case 21

Case 21: This case ignores all characters read until the \* character is read indicating the end of the multiline comment.

### Case 22

Case 22: This case handles the end of the comment. If the character is \*, the state is set to 22 looping until another character is found. If the / character is found, then the end of comment is declared. Otherwise, the \* found was part of the comment and state is set to 21.

### Case 23

Case 23 handles one-line comments. The if statement checks for / slash. If found, state is set to 23 looping until a new line starts. After the comment ends, the lexical ignores it completely.

### Case 24

This case corresponds to the character ". It is responsible for handling the start of a string literal. It iterates to itself where the subsequent characters are processed to form a string literal token. If \\ is found in the middle of the string, the state is set to 33.

Case 25

Case 25: This case corresponds to the character '. It is responsible for handling the start of a character literal. It iterates to itself where the subsequent characters are processed to form a character literal token.

### Case 26

Case 26: This case corresponds to the character '!' inside the getNextToken function. It is responsible for handling the possibility of a logical operator. Depending on the next character, it can form logical operators such as '!', '!=', or '!='. The specific transitions and actions for this case are not shown in the provided code snippet, so further details about the exact logic and behavior associated with this case are not available.

### Case 27

Case 27: This case corresponds to an unknown character inside the getNextToken function. It is responsible for handling characters that do not match any of the predefined cases. It can perform error handling routines or handle unknown characters in a specific way. The specific transitions and actions for this case are not shown in the provided code snippet, so further details about the exact logic and behavior associated with this case are not available.

### Case 28

Case 28: This case corresponds to the character '^' inside the getNextToken function. It is responsible for handling the possibility of a bitwise operator. Depending on the next character, it can form bitwise operators such as '^' or '^='.

### Case 29

Case 29: This case corresponds to the character '<<'. It is responsible for handling the possibility of a bitwise operator. Depending on the next character, it can form bitwise operators such as '<<' or '<<='.

### Case 30

Case 30: This case corresponds to the character '>>'. It is responsible for handling the possibility of a bitwise operator. Depending on the next character, it can form bitwise operators such as '>>' or '>>='.

### Case 31

Case 31 handles (’’). if a (’) is found, then state is set to 1 and lexemeType to literal. Else, if space is found state is set to 32. Otherwise, an error.

### Case 32

Case 32 handles an empty string by setting state = 1 to start with another lexeme and setting lexemetype to 11. Otherwise, an error.

### Case 33

Case 33 handles escaped characters inside a string by returning the state back to 24.

After the switch case finishes, there are some conditions to check and handle.

* If the retrackFlag is set to true. It handles the situation when a character needs to be reprocessed. The code checks the value of the forward pointer and adjusts it accordingly. If the forward pointer is pointing to buffer1, it is moved to the second-to-last character of buffer2. If the forward pointer is pointing to buffer2, it is moved to the second-to-last character of buffer1. Otherwise, if the forward pointer is pointing to any other position, it is simply decremented by one. Finally, the retrackFlag is set to false.
* If the state is neither 1 nor 0, indicating that the tokenization process was incomplete. In such a case, it prints an error message indicating that the token is incomplete. Furthermore, it traverses the characters between the start and forward pointers and outputs them on the console. Finally, it returns the partially constructed Token object.
* Another function constructs the lexeme string by appending characters from the start pointer to the forward pointer. It iterates over the characters and handles buffer switching when encountering the end of a buffer.
* Last function assigns the appropriate TokenType to the Token object based on the lexemeType. If the lexemeType is 1, indicating either a keyword or an identifier, it checks if the lexeme is a keyword by calling the isInTable function. If it is a keyword, the Token object's type is set to "keyword", and the stringValue is set to the lexeme. If the lexeme is an already existing identifier, the Token object's intValue is set to the corresponding index. If it is a new identifier, the lexeme is added to the symbolTable, and the Token object's intValue is set to STindex++. For lexemeType values between 2 and 5, representing numbers, the Token object's intValue is set by converting the lexeme string to an integer using stoi(). For lexemeType values between 6 and 12, representing other types of tokens, the Token object's stringValue is set to the lexeme string. Finally, the constructed Token object is returned.

## Last 2 functions

int isInTable(string identifier): This function checks if a given identifier exists in the symbolTable. It iterates through the elements of the symbolTable vector using a while loop. Inside the loop, it compares each entry of symbolTable with the provided identifier. If a match is found, the loop is terminated using the break statement, and the index of the matching entry is returned. If no match is found, the function returns the size of the symbolTable, indicating that the identifier is not present.

void printSymbolTable(): This function prints the contents of the symbolTable vector. It starts by printing a header with the labels "Entry" and "Identifier" to provide a clear representation. Then, it iterates through the elements of the symbolTable vector using a for loop. The loop begins from index 32, which is the starting index for user-defined identifiers (assuming the first 32 entries are reserved for keywords). For each index, it prints the index value followed by a tab character and then the corresponding identifier from the symbolTable vector. This function is used to display the entries in the symbol table after tokenization or during debugging.

The isInTable function takes a string parameter identifier, which represents the identifier to be checked. It returns an integer value indicating the result of the lookup.

The function iterates over the symbol table entries using a while loop. It compares each entry's first element (keyword or identifier) with the provided identifier. If a match is found, the function returns -1 to indicate that the identifier exists in the symbol table.

If no match is found, the function continues to the next entry until it reaches an empty string as the first element of an entry. At this point, it returns the index of the symbol table entry, which represents the position where the identifier could be added to the symbol table.